1988

Semantics of a Real-Time Language

Berzins, Valdis

IEEE

Semantics of a Real-Time Language

Valdis Berzins

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

This paper describes the semantics of the real-time language PSDL, which was designed for prototyping real-time systems.[1] We focus on the aspects of the language relating to hard real-time constraints, scheduling, and functional behavior. The main contributions of the paper are to present the aspects of PSDL that simplify the description of real-time constraints and to clarify the relationships between periodic and data driven operators and the interactions between timing and control constraints in a language that combines these features.

1. Introduction

PSDL is a language designed for clarifying the requirements of complex real-time systems, and for determining properties of proposed designs for such systems by means of prototype execution. The language was designed to simplify the description of such systems [10] and to support a prototyping method that relies on a novel decomposition criterion [8]. PSDL is also the basis for a computer-aided prototyping system [9] that speeds up the prototyping process by exploiting reusable software components [7] and providing execution support [5, 11, 14] for high level constructs appropriate for describing large real-time systems in terms of an appropriate set of abstractions [1].

Real-time systems are difficult to specify and design. PSDL alleviates these problems by providing a simple high-level view of real-time systems. This paper describes the semantics of the aspects of PSDL relevant to real-time constraints, scheduling, and the resulting functional behavior. The semantics of the language are described and illustrated with examples. The main contributions of the paper are to clarify the meaning of PSDL and to illuminate the relationships between periodic and data driven operators and the interactions between timing and control constraints. We also describe some implications for constructing schedulers for high level real-time languages, and indicate some improvements to the current version of the language.

Timing constraints for systems modeled as finite state machines have been classified as maximum, minimum, and durational [2]. PSDL has facilities for expressing these types of constraints. PSDL describes software systems as networks of operators connected by data streams, and subject to timing and control constraints. Operators can be either periodic or triggered by the sporadic arrival of input data. A dataflow model of real-time systems where input data arrive only at fixed rates has been studied in [13]. Timing requirements affecting the safety of software systems must be specified before they can be verified. This paper addresses the problem of specifying timing requirements in a way that allows the generation of an executable prototype. The verification of safety assertions involving timing constraints by means of RTL (real-time logic) is discussed in [4]. The PSDL syntax integrated with partial graphical notation is much easier for designers to use compared to temporal logic, although the underlying implementation must be in a formal form for mechanical or automated processing. The PSDL facilities for expressing timing constraints are simpler than temporal logic because they are more restricted and are closer to the concerns of the system designer.

An important goal of PSDL is to simplify the design of systems with hard real-time constraints. The need for meeting real-time deadlines often results in designs where code for conceptually unrelated tasks must be interleaved, complicating the design of such systems, and making their implementations hard to understand [3]. PSDL handles this problem by presenting a high-level description in terms of networks of independent operators, and allowing the interleaving of the code to be handled by an automatic translator that generates lower level code. High-level synchronization is handled by using dataflow streams to coordinate the arrival of corresponding data values from different sources. Static scheduling of time-critical operators [6] eliminates the need for other kinds of explicit synchronization by the system designer. Low-level synchronization primitives are needed in the implementation of PSDL only for ensuring that the read and write operations on data streams are performed as atomic operations. Since those primitives are needed in just one small and simple part of the code in the PSDL execution support system, PSDL can be effectively supported by any of the common mutual exclusion mechanisms provided by operating systems.

2. The PSDL Computational Model

The PSDL language is based on a computation model which treats software systems as networks of operators communicating via data streams. The computational model is an augmented directed graph

$$G = (V, E, T(v), C(v))$$

where V is the set of vertices, E is the set of edges, T(v) is the set of timing constraints for each vertex v, and C(v) is the set of control constraints for each vertex v. Each vertex is an **operator** and each edge is a **data path.** Each of the four components of the graph are described in more detail below. The semantics of a PSDL system description is determined by the associated augmented graph and the semantics of the operators appearing in the diagram.

2.1. Operators

All PSDL operators are state machines. Some PSDL operators are functions, i.e. machines with only one state. When an operator fires, it reads one data value from each of its input streams, undergoes a state transition, and writes at most one data value into each of its output streams. The output values can depend only on the current set of input values and the current state of the operator. State transitions and input/output operations on data streams can occur only when the associated operator fires. The firing of an operator is controlled by the associated timing and control constraints. Operators can be triggered by the arrival of a set of input data values or by a periodic temporal event.

2.2. Data Streams

PSDL operators communicate by means of named **data streams.** All PSDL data streams can carry both normal data values and tokens representing exceptions. All of the normal data values carried by a stream must be instances of a specified abstract data type associated with the stream.

A data path connects exactly one producer operator to exactly one

[1]This research was supported in part by the National Science Foundation under grant number CER-8710737.

consumer operator, corresponding to a single edge in the graph forming the computational model. A data stream connects one or more producer operators to a single consumer operator, corresponding to a nonempty set of data paths, all of which have the same name and consumer node. The producers write data values into the stream while the consumer reads data values from the stream.

As a notational convenience, PSDL allows the system designer to describe networks where several operators read values from the same data stream s. This is an abbreviation for an expanded network containing a distinct data stream s[i] for each consumer operation c[i], where the index i ranges from one to the number of consumers. Each data stream s[i] in the expanded network is a copy of the original stream s. The PSDL language translator provides implicit **copy** operators which replicate the data values written into the data stream s and distribute them to the data streams s[i]. The behavior of a binary copy operator is defined in Fig. 1. The copy operator defines the behavior of data streams with multiple consumers in terms of data streams with a single consumer. The following discussion assumes all data streams have just a single consumer.

There are two different kinds of data streams in PSDL, **dataflow** streams and **sampled** streams. Dataflow streams are used in applications where the values in the stream must not be lost or replicated and the firing rates of the producers and consumer are the same, while sampled streams are used in applications where a value must be available at all times and values can be replicated without affecting their meaning.

The behavior of each type of data stream is defined in terms of the input and output histories of the stream. All read and write operations on the same data stream are mutually exclusive in time, and occur in a well-defined order. For each stream s and time t let the input history in(s, t) be the sequence of data values that were written into the stream s from the time the system started operation up to time t, appearing in the order in which they were written into the stream. Let the output history out(s, t) be the sequence of data values that were read from the stream s from the time the system started operation up to time t, appearing in the order in which they were read from the stream. We will write input and output histories so that the oldest data elements appear on the left and the most recent ones appear on the right.

If s is a dataflow stream then the relation between the input history of s and the output history of s is described by the following formula in ordinary first order logic,

All t: time, s': sequence [in(s, t) = (out(s, t) || s') & length(s') <= 1]

where "||" denotes sequence concatenation. The formula says the output history of the stream is always a prefix of the input history, and that the length of the two sequences can differ by at most one. The sequence of data values s' represents the contents of the buffer implementing the data stream at the time t, since it contains the data values that have been written into the data stream but not yet read out. Consequently, dataflow streams can be implemented as FIFO queues of length one.

Dataflow streams guarantee that each of the data values written into the stream is read exactly once, unless the entire computation terminates with a non-empty queue for some stream. Computation sequences that would require a value to be written into a full queue or to be read from an empty queue are invalid, and result in an error message. Fig. 2 illustrates the relation between the input history of a dataflow stream and the output history of the stream. Part (a) shows the sequence of values written into the integer stream s up to the time t, and (b) and (c) show the two possible sequences of values that may be read from the stream s up to the time t. All other sequences are illegal output histories for the stream s up to the time t.

If s is a sampled stream then the sequence of distinct values read from a sampled stream is a subsequence of the input history of the stream. This allows some of the values written into the stream to be lost or replicated if

```
OPERATOR copy
  SPECIFICATION
    GENERIC t: type
    INPUT x: t
    OUTPUT y, z: t
    AXIOMS { y = x & z = x }
END
```

Fig. 1 Copy Operator

(a) in(s, t) = [1, 2]

(b) out(s, t) = [1] (legal)

(c) out(s, t) = [1, 2] (legal)

Fig. 2 Behavior of a Dataflow Stream

that is required to meet timing constraints, as is commonly necessary in asynchronous systems. More precisely, a sampled stream s satisfies the relation

All t: time [subsequence(distinct(out(s, t)), in(s, t))]

where the relation "subsequence" is defined by the following properties:

subsequence([ ], s) = true
subsequence([x] || s, [ ]) = false
subsequence([x] || s, [y] || s') =
  if x = y then subsequence(s, s') else subsequence([x] || s, s')

and the function "distinct" is defined by the following properties:

if length(s) <= 1 then distinct(s) = s
distinct([x, y] || s) =
  if x = y then distinct([x] || s) else [x] || distinct([y] || s)

Subsequence(s1, s2) is true if the elements of the sequence s1 are embedded in the sequence s2 in the same order, corresponding to standard mathematical usage. For example, subsequence([1, 3], [1, 2, 3]) = true. Distinct(s) is a sequence containing the same data values as the sequence s, in the same order, where each block of adjacent identical data values has been replaced by a single copy of the data value. For example, distinct([1, 1, 1, 2, 2]) = [1, 2].

Fig. 3 illustrates the relationship between the values written into a sampled stream and the values read from the stream. Part (a) shows the sequence of values that have been written into an integer data stream up to a particular time t, while (b) and (c) show possible sequences of values read from the stream up to time t. In (b) input values have been lost and in (c) they have been replicated. Part (d) shows a sequence of values that could not be read from the sequence up to time t, because the order of the values is not preserved. Sampled streams can be implemented by a memory cell holding the value most recently written into the stream. Computation sequences that would require a value to be read from an uninitialized sampled stream are invalid, and result in an error message. Errors of this type can be avoided by declaring initial values for streams.

Both kinds of data streams preserve the order of the data values they carry. This means data values cannot pass each other in a data stream. More precisely, if the value x1 was read from the stream s before the value x2 was read from s, then it is impossible for x1 to have been written into s later than x2.

2.3. Timing Constraints

Any PSDL operator can have timing constraints associated with it. An operator is **time-critical** if it has at least one timing constraint associated with it, and is **non time-critical** otherwise. The timing constraints together with the control constraints determine when the operator can be fired, and when it must be fired. All PSDL timing constraints can be represented by constants denoting lengths of time intervals. There are several different kinds of timing constraints, which can be classified into those that apply to all time-critical operators, those that apply only to operators triggered by periodic temporal events, and those that apply only to operators triggered by the arrival of new data. Temporal events occur at

(a) in(s, t) = [1, 2]

(b) out(s, t) = [2] (legal)

(c) out(s, t) = [1, 1, 1, 2, 2] (legal)

(d) out(s, t) = [2, 1] (illegal)

Fig. 3 Behavior of a Sampled Stream

specified absolute times.

Every time-critical operator must have a **maximum execution time** (MET) to allow the construction of a static schedule. The MET of an operator is an upper bound on the length of the **execution interval** (EI) for the operator. All of the actions that may be required to fire an operator once must fit into the execution interval. These actions are listed below.

(1) Reading values from input data streams.

(2) Evaluating triggering conditions.

(3) Calculating output values.

(4) Evaluating output guards.

(5) Writing values into output streams.

The execution interval for an operator does not include scheduling delays. A scheduling delay is the time between the writing of a value into a data stream by a producer operator and the reading of that value by the consumer operator.

Operators triggered by temporal events are periodic in PSDL. Every periodic operator must have a **period** (PERIOD) and may have a **deadline** (FINISH_WITHIN). These two timing constraints partially determine the set of **scheduling intervals** (SI) for the operator. Each periodic operator must be fired exactly once in each scheduling interval, and must complete execution before the end of the scheduling interval. The period is the length of time between the start of any scheduling interval and the start of the next scheduling interval. The deadline is the length of each scheduling interval. The starting time of the first scheduling interval for each operator is determined by the static scheduler, subject to the scheduling constraints described in section 2.4.

The relation between the timing constraints, scheduling intervals, and execution intervals for a periodic operator is illustrated in Fig. 4. The execution intervals and scheduling intervals in the diagram are indexed by integers in the order of their occurrence. Thus SI[n] denotes the n-th scheduling interval for the operator and EI[n] denotes the n-th execution interval for the operator. The static scheduler takes the length of each execution interval to be equal to the maximum execution time to allow for worst case conditions. If a time-critical operator completes before the end of the execution interval reserved for it by the static scheduler, the remaining time in the execution interval is used by the dynamic scheduler for the execution of a non time-critical operator.

Since each execution interval must be a subset of the corresponding scheduling interval, every well-formed periodic operator must satisfy the constraint

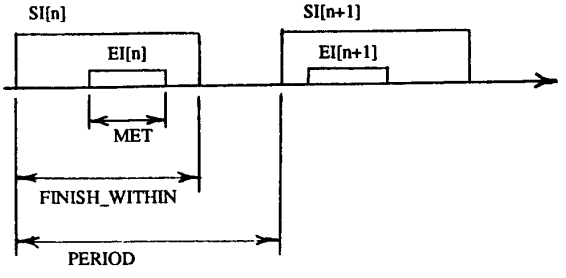
MET <= FINISH_WITHIN.

The degree of freedom enjoyed by the static scheduler is characterized by the slack, which is defined by

slack = FINISH_WITHIN - MET.

The length of time between the start of an execution interval and the start of the next execution interval can vary between

PERIOD - slack



SI[n] = n-th scheduling interval
EI[n] = n-th execution interval

Fig. 4 Timing Constraints for a Periodic Operator

and

PERIOD + slack,

as illustrated in Fig. 5. If FINISH_WITHIN is not specified explicitly then

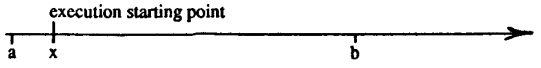
FINISH_WITHIN = MET,
slack = 0,

and the time between the starting points of each pair of consecutive execution intervals is exactly equal to the period.

Operators triggered by the arrival of new data values are sporadic. Timing constraints for sporadic operators are optional. Sporadic operators with timing constraints must have both a **maximum response time** (MRT) and a **minimum calling period** (MCP) in addition to an MET. The MRT is an upper bound on the **response time**, while the MCP is a lower bound on the **calling period**. The relation between these quantities is illustrated in Fig. 6. SI[n] denotes the n-th scheduling interval for the consumer operator, which is sporadic and time-critical. CEI[n] denotes the n-th execution interval for the consumer operator, and PEI[n] denotes the n-th execution interval for the producer operator, which is assumed here to be time-critical also. The response time associated with a consumer operator is measured from the end of the execution interval for the producer operator of the triggering data value to the end of the execution interval for the consumer operator of the triggering data value. In cases where the consumer operator is triggered by a set of several data values (see TRIGGERED BY ALL, section 2.4), the triggering data value is the element of that set that was produced last.

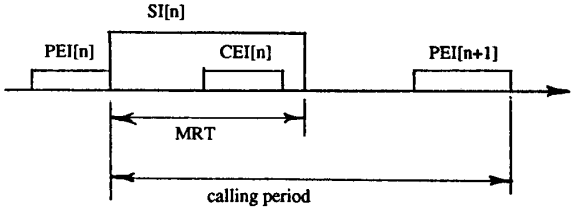
Unlike the MET, the MRT includes a scheduling delay. The MRT gives the length of the scheduling interval. As discussed in section 3, the static scheduler may not be able to use the entire scheduling interval if the producer is non time-critical, because the ending time of the producer's execution interval is not known to the static scheduler in that case.

The calling period of an operator is the length of time between the end of the execution interval for the producer of the triggering data value and the end of the execution interval for the producer of the next triggering data value. The calling period must not exceed the MCP. The MCP of an operator constrains the behavior of the producers of the triggering data values rather than constraining the behavior of the operator itself. An MCP constraint is needed to allow the realization of a maximum response time constraint with a fixed amount of computational resources, via a limit on the frequency with which new data can arrive. Violation of an MCP constraint should result in a warning message. The absence of such violations can be



slack = FINISH_WITHIN - MET
a = period - slack
b = period + slack
a <= x <= b

Fig. 5 Scheduling Freedom for a Periodic Operator



PEI[n] = n-th producer execution interval
CEI[n] = n-th consumer execution interval
SI[n] = n-th scheduling interval

Fig. 6 Timing Constraints for a Sporadic Operator

guaranteed a priori only if the producer of each triggering data value is scheduled statically. An important case where such a guarantee is not possible occurs when the producer of the triggering data value is an asynchronous active sensor.

2.4. Control Constraints

PSDL operators are subject to control constraints, which are used for the following purposes:

(1) Controlling operator execution.

(2) Controlling output.

(3) Controlling exceptions.

(4) Controlling timers.

Exceptions and timers are beyond the scope of this paper, and are discussed in [7].

Control constraints controlling operator execution are called **triggering conditions**. The forms of PSDL triggering conditions are shown in Fig. 7. A triggering condition has two parts, the **trigger** and the **guard**, both of which are optional. The trigger defines the conditions under which an operator can be fired. The keywords "TRIGGERED BY ALL" indicate the operator can be fired if new data values are present in all of the input data streams named in the id_list. The id_list can contain any non-empty subset of the input data streams for the operator. A data stream has a new data value if there has been a write operation on the stream after the most recent read operation on the stream. The new data value can be equal to the previous data value if the arguments of two consecutive write operations are the same. The natural dataflow firing rule corresponds to a "TRIGGERED BY ALL" triggering condition that lists all of the input data streams of the operator.

The keywords "TRIGGERED BY SOME" indicate the operator can be fired if there is a new data value on at least one of the data streams named in the id_list. A null trigger is equivalent to a "TRIGGERED BY SOME" that lists all the input data streams of the operator.

The triggering conditions of the operators implicitly determine the types of the data streams. A stream is a dataflow stream if it appears in a "TRIGGERED BY ALL" constraint of the consumer operator and is a sampled stream otherwise.

The conditions under which an operator can be fired are defined above. The conditions under which an operator must be fired depend on the timing constraints associated with the operator. Periodic operators must be fired once in each period in which the new data values required by the triggering condition are available. Sporadic operators with a maximum response time must be fired within the specified time interval, which starts at the instant all of the new data values required by the triggering condition become available. Sporadic operators without a maximum response time must be fired after some finite but potentially unbounded delay if the input data required by the triggering condition is available and the output data streams are not full. One consequence of the rule for non time-critical sporadic operators is that the data in sampled input streams of such an operator can be overwritten several times before the operator fires. This can also occur if a producer and the consumer are periodic and the producer has a shorter period. Such behavior is considered normal for sampled streams, and no errors are reported.

Delays for non time-critical operators can be due to lack of spare processor time or to full output streams. Delays due to full output streams can only be caused by dataflow streams produced by sporadic operators without timing constraints. Sampled streams are never full because the value in a sampled stream can be overwritten at any time. Operators with timing constraints must fire at specified times, and if the required input data is not available or if there is no room for the output data, the result is a stream error.

The second part of the triggering condition is the guard, which is a boolean expression depending on the input values and locally available state information. A null guard is always true. When an operator fires, it reads one data value from each of its input streams. If the guard is true for these values then the output values of the operator are calculated, and otherwise the firing of the operator terminates immediately without producing any output. In particular, if the guard is false the operator cannot raise any exception conditions or perform any diverging computations.

```
triggering_condition =
   "OPERATOR" id "TRIGGERED" ["BY" trigger] ["IF" guard]
trigger = "ALL" id_list | "SOME" id_list
```

Fig. 7 PSDL Triggering Conditions

Constraints for controlling output are called **output guards**. There can be an output guard associated with each output data stream. An output guard is a boolean expression that can depend on the input data of the operator, locally available state information, and the calculated output values. A null output guard is always true. An output value is written into an output stream of an operator if the operator is triggered, calculates its output values, and the output guard associated with the stream is true. If the output guard associted with a stream is false, then nothing is written into the stream. Output guards can be used to selectively disable some of the outputs of an operator.

2.5. Scheduling Constraints

There are two implicit constraints on the order in which PSDL operators can be scheduled to fire, the dataflow precedence constraint and the non-interference constraint. The dataflow precedence constraint requires the initial firings of all operators with timing constraints to occur in an order consistent with the **dataflow ordering**, which is defined in terms of the PSDL computational model as follows. Construct the **precedence graph** by taking all of the nodes in the augmented graph G defined at the beginning of section 2, and taking only the edges of G that do not have explicitly declared initial values. Since any cycle of G must contain at least one edge with a declared initial value in a well formed PSDL program, the precedence graph is acyclic. The dataflow ordering is the transitive closure of the precedence graph, which results in a strict partial ordering. For any pair of operators (a, b), if a precedes b in the dataflow ordering then a must be scheduled to fire before b is scheduled to fire for the first time.

Fig. 8 illustrates the scheduling constraints imposed by the dataflow ordering. Part (a) shows an augmented graph, (b) shows the corresponding precedence graph, assuming the data stream w has a declared initial value, and (c) shows an initial segment of a two processor schedule consistent with the dataflow ordering. All of the operators in the example are assumed to be periodic with a period of at least 8 time units.

The non-interference constraint requires all operators to be scheduled so that the sequence of output values produced by each operator is the same as the one produced by a schedule in which each operator receives the same sequence of input values and no two execution intervals of the same operator overlap in time. This constraint prevents output values from getting out of order and prevents interference between two different transactions at the same state machine. These properties are important because they allow the unification of control flow and data flow decomposition criteria [8]. No res-

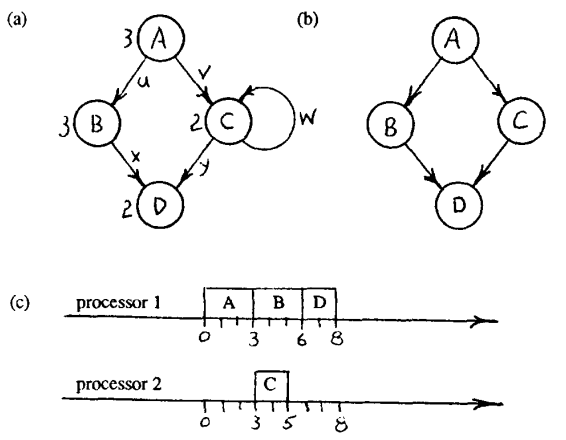


Fig. 8 The Dataflow Scheduling Constraint

triction on the overlap of execution intervals is needed for distinct operators because states of distinct operators are independent and all data references in PSDL are strictly localized. The non-interference constraint is similar to the serializability property used to define concurrency control requirements for databases.

3. Scheduling Implications

The timing constraints for a sporadic operator can be specified in the same way regardless of whether the producer of the triggering data value is subject to timing constraints, but the scheduling problem differs for the two cases. The usual way to guarantee each operator will receive enough processor time to meet its deadlines in all cases is to statically allocate time slots long enough to accommodate the worst case execution time of each time-critical operator. In this approach, all time-critical operators appear in a static schedule, and all non time-critical operators are scheduled dynamically in time slots that are not needed for the execution of time-critical operators. If the producer of the triggering data value has PSDL timing constraints, then it will appear in the static schedule, so that the start of the scheduling interval for the sporadic operator can be determined exactly when the schedule is constructed (see Fig. 6). If the producer of the triggering data value does not have PSDL timing constraints, then the exact arrival time of the triggering data value is not known at the time the static schedule is constructed, and the schedule must be based on a range of possible arrival times. Such an operator can be scheduled as if it were periodic, with an equivalent period P given by

$$P = \min(MCP, MRT - MET)$$

and a deadline equal to MET [12]. We suggest the following set of refinements to this well known approach. These refinements sometimes allow valid schedules to be constructed where the basic approach would not find a valid schedule, at the expense of complicating the construction of the static schedule.

(1) Apply the equivalent period only if the producer of the triggering data value is not in the static schedule.

(2) Treat P as an upper bound on the period between execution intervals, rather than an exact specification of the equivalent period. The static schedule may contain areas of congestion due to other periodic operators, which can sometimes be bridged by scheduling the sporadic operator at shorter intervals just before the congested time period. This strategy works by adjusting the phase of the equivalent period to match the end of an execution interval for the sporadic operator to the beginning of the congested period in the static schedule, thereby allowing most effective use of a bridging span of length P.

(3) Using multiple processors in cases where P < MET if execution intervals of the sporadic operator can be allowed to overlap without violating the second scheduling constraint defined in section 2.5. One case where this is possible is for operators without internal states for which

$$\text{(maximum execution time)} - \text{(minimum execution time)} < P/N$$

where N is the number of processors used. This restriction is needed to make sure output values do not get out of order. The strategy can be made more widely applicable by artificially adding delays to ensure each operator writes its results into the output data streams only at the end of an execution interval of length MET.

The first scheduling constraint of section 2.5 has sometimes been treated by performing a topological sort of the operators with respect to the dataflow ordering. This is appropriate for single processor implementations. In multiple processor implementations, however, undetermined aspects of the dataflow ordering can be exploited, as illustrated in Fig. 8.

4. Conclusions

The PSDL language provides a set of high level constructs for describing systems with real-time constraints. This paper provides a precise but informal description of the aspects of the language relevant to real-time constraints and scheduling. We have also traced the interactions between the real-time constraints, and the control and scheduling constraints that determine the functional behavior of a real-time program.

The study reported here has identified some additional kinds of timing constraints that would be useful additions to the PSDL language. These include the following.

(1) A maximum calling period. This kind of constraint is useful for describing systems with a minimum refresh rate, such as displays or dynamic memories. Such a constraint should be linked to a default data value to be supplied in case new data does not arrive within the specified timeout period. Currently such constraints can be expressed using timers and extra operators, but such a description contains details that are not logically necessary.

(2) A minimum response time. This kind of constraint is useful for driving external systems with maximum data rates, such as communications channels, or sporadic operators with specified minimum calling periods.

More research is needed to determine ways to effectively schedule operators on multiple processors. Some specific research questions identified in this paper are to determine easily decidable conditions under which scheduling algorithms can safely schedule two different firings of the same operator in overlapping time intervals, and methods for statically determining the absence of potential error conditions.

1. V. Berzins, "The Design of Software Interfaces in Spec", in *to appear in Proceedings of the International Conference on Computer Languages*, Miami, Oct. 1988.
2. B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them", in *Proceedings of the Real-Time Systems Symposium*, IEEE, New Orleans, LA, Dec. 1982, 197-204.
3. S. Faulk and D. Parnas, "On Synchronization in Hard-Real-Time Systems", *Comm. of the ACM 31*, 3 (Mar. 1988), 274-187.
4. F. Jahanian and A. Mok, "Safety Analysis of Timing Properties in Real-Time Systems", *IEEE Trans. on Software Eng. SE-12*, 9 (Sep. 1986), 890-904.
5. D. Janson, "A static Scheduler for Hard Real-Time Constraints in the Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.
6. D. Janson and Luqi, "A Static Scheduler for the Computer Aided Prototyping System", in *to appear in Proceedings of COMPASS 88*, Gaithersburg, MD, June 1988.
7. Luqi, "Rapid Prototyping for Large Software System Design", Ph. D. Thesis, University of Minnesota, 1986.
8. Luqi and V. Berzins, "Rapid Prototyping of Real-Time Systems", *IEEE Software*, Sep. 1988.
9. Luqi and M. Ketabchi, "A Computer Aided Prototyping System", *IEEE Software*, March 1988.
10. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. on Software Eng.*, October, 1988.
11. C. Moffitt, "Development of a Language Translator for a Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.
12. A. K. Mok, "The Decomposition of Real-Time System Requirements into Process Models", *IEEE Proc. of the 1984 Real Time Systems Symposium*, Dec. 1984, 125-133.
13. A. Mok and S. Sutanthavibul, "Modeling and Scheduling of Dataflow Real-Time Systems", in *Proceedings of the Real-Time Systems Symposium*, IEEE, San Diego, CA, Dec. 1985, 178-187.
14. J. O'Hem, "A Conceptual Design of a Static Scheduler for Hard Real-Time Systems", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.